Fuzzing with AFL

Michael Macnair
@michael_macnair

GrayHat, October 2020

Agenda

- Intro to fuzzing and AFL
- *Exercise – fuzz a toy program*
- *Exercise - understanding test harnesses*
- Practical fuzzing with AFL
- *Challenges*
- Tips, target selection, limitations
- Resources
- *Challenges continue*

Not Covering

- History of fuzzing
- Fuzzing without source
- Fuzzing over a network
- Target specific techniques
- Architecturally complex targets (e.g. kernel)
- Any tools other than afl (and a bit of libFuzzer)
- Running at scale
- Crash triage
- Current research

Background (1)

- Software: f(input) -> output
- Applies at any scale: PowerPoint, grep, the Linux kernel, a library, a class, a function, a snippet of code.
- Sometimes, f(unexpected_input) -> security flaw

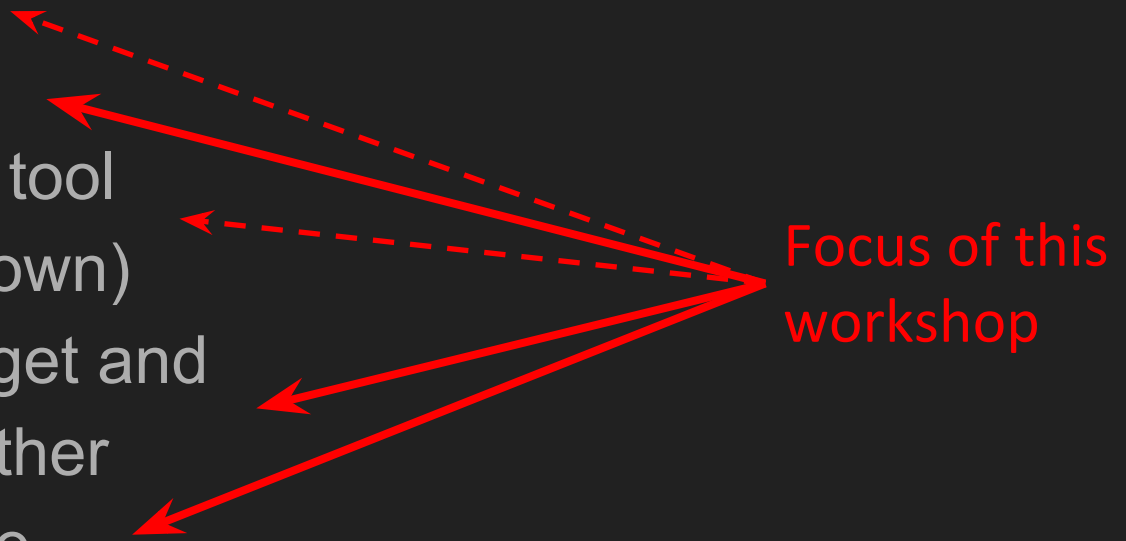
Background (2)

- Finding if there are any inputs that cause security issues is difficult.
  - Design review, code review, static analysis, dynamic analysis, …
- Dynamic analysis: run program on input, check it doesn’t violate your constraints.
- Fuzzing
  - Simplest constraint: doesn’t crash.
  - Random inputs
- Surprisingly effective!

Background (3)

Fuzzing steps:

1. Pick a target
2. Identify inputs
3. Pick a fuzzing tool (or write your own)
4. Make your target and tool work together
5. Fuzz until done
6. Triage the results ✖



What fuzzing is good for

- Finding crashes that can be triggered by input
- Running *lots* of tests that you never would
- Examples:
  - PDF readers; graphics; office suites; ..
  - libraries (images; audio; ...)
  - perl, clang, gcc, sqlite, ...
  - browsers
  - Unix tools: strings, file,
  - Crypto APIs, filesystem drivers, …
  - OS kernels

American Fuzzy Lop

> *a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.*

Ubiquitously known as ‘afl’

The original version is no longer maintained: we now use afl++

Why afl?

- Joint best general purpose fuzzer
  - For C/C++/Objective C programs
- Effective
  - Fast – lots of test cases per second
  - Clever – test cases are well chosen
- Easiest to use

- (see also: libFuzzer)

What does afl do?

- Coverage guided fuzzer
- Compiler instruments code
- AFL learns code path taken for a given input
- Genetic algorithm - mutates inputs that hit new code paths*
- * technically new tuples of basicblock-basicblock
- Skips entries that provide a subset of coverage
- Probabilistically skips unfavoured entries
  - Slow / large
- Various mutation strategies - deterministic + random
- Dictionaries to help with magic values
- Pragmatic:

  The tool can be thought of as a collection of hacks that have been tested in practice, found to be surprisingly effective, and have been implemented in the simplest, most robust way I could think of at the time.

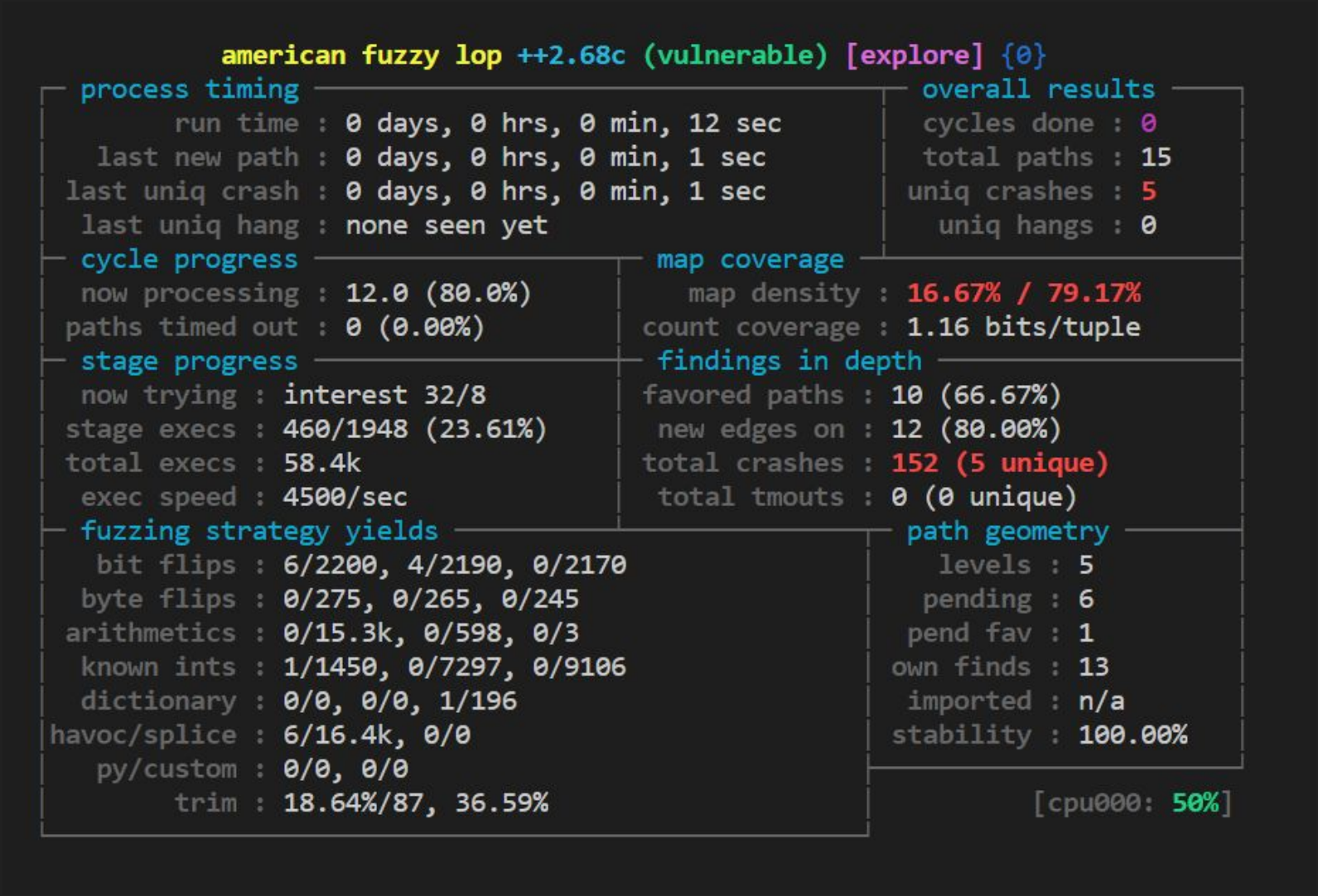
Workshop Environment

- Your own VM, running the Docker container
- Edit remotely:
  - Terminal – **nano**, vim, emacs
  - Visual Studio Code – Remote SSH
- Or edit locally, sync to remote:
  - scp/rsync
  - git (fork the repo, push/pull)
- Alternatives:
  - Local docker:
    docker run -it -u fuzzer --privileged mykter/afl-training /bin/bash
  - On host:
    git clone https://github.com/mykter/afl-training.git

Practical 1

- Taking afl for a spin
- `ssh fuzzer@<IP>`
- IP & password: check your DMs!
- Instructions in `workshop/quickstart/README.md`

Status screen



Practical 1 - quickstart

- Walkthrough
- What did afl find?
- Why?
- What if you didn’t have the `head` seed?
- How might we find the easter egg?

Test Harnesses

- Create a harness that will:
  - Run in the foreground
  - Usefully process input on stdin (or a specified file)
    - Feed input to your target: whole program (maybe you just modified its main function), or specific API, or function, or code snippet
  - Exit cleanly
  - Crash on an error, if going beyond crashes
  - (skip checksums + cryptographic integrity tests)
  - Use afl's deferred forkserver + persistent mode

Practical 2 - Test Harness

- workshop/harness/README.md
- Create a fuzz harness for a library

Documentation

- AFLplusplus/docs/
- In particular
  - docs/life_pro_tips.txt
  - docs/README.md
  - llvm_mode/README.md

LLVM Mode

- Deferred forkserver: tell afl where to start each new process
  - Especially useful for processes with an expensive setup phase that isn't dependent on input
- Persistent mode: don't fork for each run, just loop around this bit of code
  - Especially useful for fast targets
- llvm_mode/README.md

Initial Test Cases

- (aka seed corpus)
- Don’t waste core-weeks trying to synthesise your target’s input format (like this!)
- Find some real inputs that exercise as much of the target as possible.
- From the README:
  - Keep the files small. Under 1 kB is ideal, although not strictly necessary.
  - Use multiple test cases only if they are functionally different from each other. There is no point in using fifty different vacation photos to fuzz an image library.

Sanitizers & Hardening

- Spot things you otherwise wouldn't
  - Inputs that lead the code to do bad things that don't crash
- UBSan, MSan, ASan, TSan
- AFL_HARDEN=1
- Mutually incompatible
  - Run at least one fuzzer with each sanitiser, and potentially with the different options AFL supports, see the guidance in the afl++ README

Parallel Fuzzing

- Not as frictionless as the rest of afl
- Necessary for high performance
- Multicore
  - One main (`-M`) instance, one secondary (`-S`) per core. All share the same output directory (-o).
- Multi machine
  - Lots and lots of secondary instances
  - Script to sync state directories periodically
- Run fuzzers with differing behaviours (sanitizers; fuzzing algorithms)
- At a certain point, system call overheads often cap the performance gains from multi-core fuzzing

Dictionaries

- `afl-fuzz -x my.dict`
- Help the fuzzer to access paths it otherwise wouldn't
- Get them from:
  - `dictionaries` folder
  - Auto dictionary with afl-clang-lto (llvm11+)
  - libFuzzer's collection
  - libtokencap
    - subdir in afl, creates a dictionary by intercepting calls like strcmp and memcmp.
    - Useful for identifying `head` in quickstart!
  - Source code review (/grep)

Practical 3 - Challenges

- Recommended challenge order
  1. libxml2
  2. heartbleed
  3. sendmail/1301
  4. ntpq
  5. date
  6. cyber-grand-challenge
  7. sendmail/1305

- Go!

- Regardless of what you attempt, do read all the README.md, HINTS.md, and ANSWERS.md files for the different challenges

Beyond memory corruption

- Add asserts to detect bad things, e.g.:
  - Send input into two different math/crypto libraries and asserts that their output is identical
  - assert that result of BN_sqr(x) == BN_mul(x,x) - CVE-2014-3570 in OpenSSL
  - assert if any filesystem changes have occurred – CVE-2014-6271 in Bash (shellshock)

Misc Tips

- Network targets
  - Ideally write a harness around the target function (e.g. the parser)
  - Use inetd mode if your target supports it
  - Intercept network system calls, e.g. using preeny
- Disable checksums
- The `queue` directory contains a corpus of test cases that exercise your program
  - Run it through CoverageSanitizer or gcov – see what isn't hit
  - Integrate into CI, check for no crashes
  - More?
- Resume a fuzzing run with `-i-` (robust, not quick)

Triage Tips

- ASan traces can be helpful
- `afl-tmin` to minimise and simplify test cases whilst retaining the original control flow
- “Unique” bugs are not unique! Fix ones you can and repeat.
- Memory limits can give false positives
- Scripting helps with lots of crashes (collect stdout/stderr; run with different sanitizers; get a gdb backtrace; …)

When To Stop

- Never? Fuzz as part of continuous integration
- When the cycles counter is green
  - Last new path was found many cycles ago
  - Pending paths is zero
- If you want to stop earlier:
  - Cycles counter is blue (=> last new path was over a cycle ago)
  - It’s been running for a while (hours + millions of executions + at least 2 cycles)
  - Look at the output of afl-plot
- Remember to check code coverage

Limitations: fuzzing

- Hard to tell when to stop
- Tests the target in the exact configuration you provided, on the input source you set up
- Can get stuck (e.g. checksums)
- Only notices problems that can be automatically detected

Limitations: afl

- Crashes only
  - Typical of most fuzzers
- stdin / file input only
- Linux/OSX only
  - Windows options: QEMU+Wine / winafl
- Need to build target from source
  - QEMU, Unicorn, Dynamorio
- Gets stuck on magic values
  - auto dictionary / Libtokencap / laf-intel can help
- Basic-block instrumentation won't guide it towards all crashes, e.g. `x=1/(input - 1234)`
- No native parallelisation

Target Selection

- Attributes that make a good target:
  - Parsers
  - Non memory safe languages (C, C++)
  - Legacy code
  - New code
  - Complex code
  - Code with a history of flaws
  - Code that no-one has looked at before

Non-targets

- Memory safe code, that doesn't require high availability, where you aren't able to write a harness that detects 'bad' conditions
- Popular 3rd party components that have a lot of external attention (libpng, OpenSSL network facing code, protobuf's c++ implementation, etc are all probably "fuzz clean")
- Test/development/non-production components
- Components with no input?

Getting Started – Developers

- Fuzz anything suitable
  - *Every* parser written in a memory-unsafe language
- If you're writing something, write a fuzz test harness and fuzz it.
- If you're reviewing something, write a fuzz test harness and fuzz it
  - Show your team it, then next time ask them to fuzz it and show you the results
- For open source projects, write an OSS-Fuzz integration (and possibly even get paid!)

libFuzzer

- The other top general purpose C/C++ fuzzer
- Part of LLVM
- Targets functions rather than programs
  - You always have to write a harness
- In-process -> faster (no forking)
- Similar algorithms to afl

Resources

Please give me feedback (it's tiny and quick):
https://www.surveymonkey.co.uk/r/YMBP5DC
Thanks!

- The afl docs/ directory
- libFuzzer
  - libFuzzer tutorial
  - libFuzzer workshop
- Ben Nagy's "Finding Bugs in OS X using AFL" (video)
- The afl-users mailing list
- The smart fuzzer revolution (talk on the future of fuzzing): video / slides
- Papers: Fuzzing: Art, Science, and Engineering, then all the rest
- The fuzzing book - broad coverage
- More challenges from an EkoParty workshop
- Introduction to triaging crashes
- Google's ClusterFuzz and Microsoft's OneFuzz